THE GANITH ALGEBRAIC GEOMETRY TOOLKIT V1.0

Chanderjit Bajaj
Andrew V. Royappa

CSD-TR-91-065
August 1991

The GANITH Algebraic Geometry Toolkit*

Chanderjit Bajaj *Andrew V. Royappa*
Department of Computer Science
Purdue University
West Lafayette, IN 47907

Abstract

The GANITH algebraic geometry toolkit manipulates arbitrary degree polynomials and power series. It can be used to solve a system of algebraic equations and visualize its multiple solutions. Example applications of this for geometric modeling and computer graphics are curve and surface display, curve-curve intersections, surface-surface intersections, global and local parameterizations, implicitizations, and inversions. It also incorporates techniques for multivariate interpolation and least-squares approximation to an arbitrary collection of points and curves.

*Supported in part by NSF grant CCR 90-02228 and AFOSR contract 91-0276

Contents

1 INTRODUCTION **3**

2 TECHNICAL TOUR **4**
- 2.1 Functional Subsystems 4
- 2.2 Visualization Algorithms 4
  - 2.2.1 Curve and Surface Display 4
  - 2.2.2 Animation 7
- 2.3 Symbolic Computation 7
  - 2.3.1 Intersection by Birational Maps 8
  - 2.3.2 Global Parameterization 8
  - 2.3.3 Local Parameterization at Singularities 9
  - 2.3.4 Surface Fitting with Algebraic Surface Patches 9
- 2.4 System Design 9
- 2.5 Extensibility 9

3 USER'S MANUAL **10**
- 3.1 The User Interface 10
  - 3.1.1 Discretization 10
  - 3.1.2 Visualization 11
  - 3.1.3 Input/Output 11
- 3.2 The Computer Algebra Library 11
- 3.3 REFERENCE MANUAL 13
  - 3.3.1 Terminology 13
  - 3.3.2 Organization 14
  - 3.3.3 Interacting with Ganith 15
  - 3.3.4 Commands and Expressions 18
- 3.4 Enhancements 20
- 3.5 TUTORIAL 20

4 PROGRAMMER'S GUIDE **24**
- 4.1 User Interface Implementation and File Structure 24
- 4.2 Computer Algebra Library Implementation 27
- 4.3 C Side Functions 29
- 4.4 Lisp Side Functions 35
- 4.5 Extension Interfaces 39
  - 4.5.1 C Language Interface 39
  - 4.5.2 Lisp Language Interface 41
- 4.6 Building Ganith 41
  - 4.6.1 Building the User Interface 41
  - 4.6.2 Building the Lisp Binary 41

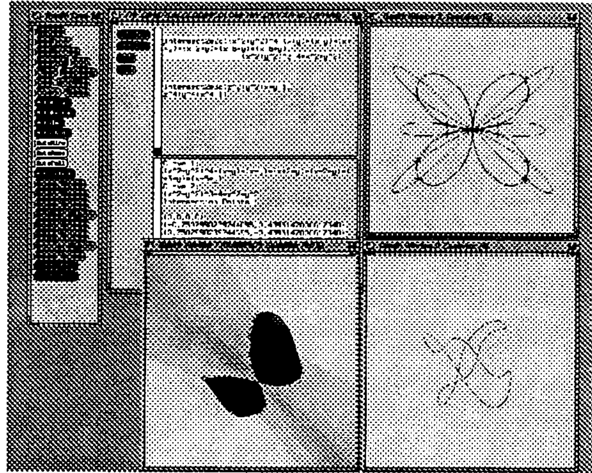

Figure 1: Display of Curve-Curve and Surface-Surface Intersections

1 INTRODUCTION

GANITH is an X-11 and XS based [6] application toolkit that manipulates polynomials and power series. Its main use is as a tool for solving systems of algebraic equations. In some special cases the user is provided with a variety of ways to display and manipulate solutions. Example applications are curve and surface display, curve-curve intersections, surface-surface intersections, global and local parameterizations, implicitizations, inversions, etc. See Figures 1 and 2.

GANITH consists of two main subsystems. The first consists of a graphical user interface and visualization system. This part is written in C. The second part, written in Lisp, is a Computer Algebra (CA) library. The two parts reside in separate Unix processes and communicate via the pipe mechanism.

The whole system is controlled by a multi-window graphical user interface system that provides convenient methods for controlling the three subsystems. GANITH as a whole is a subsystem of SHASTRA and will be integrated into the SHASTRA environment[5].

The rest of this paper consists of the following. Section 2 presents the technical details of both the software architechture, and the choice and rational of the algorithms and data structures implemented. It also highlights the new contributions made in solving systems of polynomial equations and visualizing its solutions. Section 3 may be treated as a user manual and details the functionality of the three subsystems in terms of the user interface. This section also includes an example with a tutorial. Section 4 examines the system internals and provides a programmer's guide. Finally the last section discusses future extensions and plans.

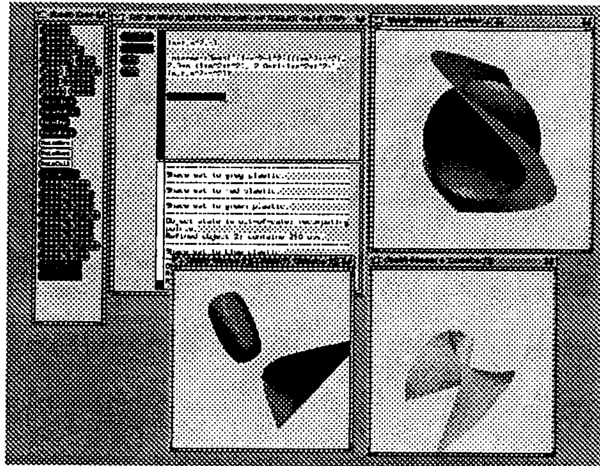

Figure 2: Shaded Display of Surfaces and Space Curves

2 TECHNICAL TOUR

2.1 Functional Subsystems

There are four major components of Ganith: user interface, controller, numerical and graphics subsystem, and algebra subsystem. The first three components reside in one process, and the algebra subsystem in another process. The algebra subsystem is written in Common Lisp, and the others in C and Fortran. All graphics and numerical calculations are performed in C or Fortran, while symbolic and exact calculations are done in Lisp. The two processes communicate with each other using Unix sockets, and may run on different hosts.

To start Ganith, a C program containing the user interface, controller, and numerical subsystem is executed. This invokes the algebra subsystem in a Lisp process, possibly on another host. The user interface is used for command input and output. Once a command is entered and edited to satisfaction, the controller sends it to the numerical or algebraic subsystems for execution, then the results are sent to the user interface for display. The algebraic and numerical subsystems may request computations from each other via the controller.

2.2 Visualization Algorithms

2.2.1 Curve and Surface Display

A number of facilities for the display and manipulation of algebraic curves and surfaces are provided. Curves and surfaces may be specified in either implicit or parametric form. The display algorithms generally start by finding a *piecewise linear approximation* (PLA)

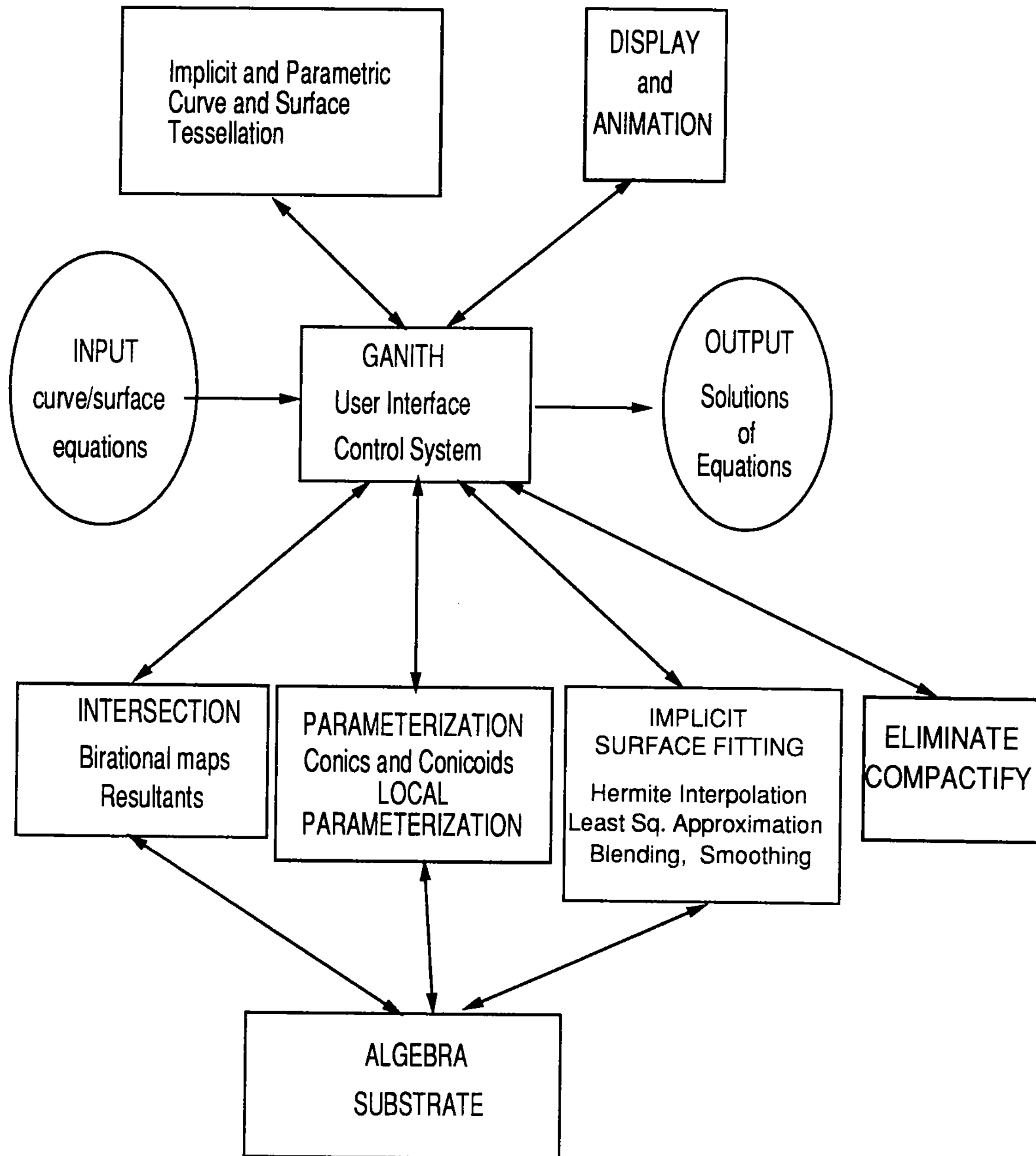

Figure 3: The GANITH Software Architecture

to a curve or surface. That is, a curve is approximated by line segments, and a surface by polygons.

- *Plane Curves in Implicit Form*
  A curve is specified by its polynomial equation $f(x,y) = 0$. A PLA is constructed using the recursive plane-subdivision technique of Geisow [25]. The resulting list of line segments is displayed. The user selects the portion of the plane over which to graph the curve. An alternative curve tracing scheme using quadratic transformations for local desingularization has also been implemented [12].

- *Surfaces in Implicit Form*
  A surface is specified by its equation $f(x,y,z) = 0$. A PLA is constructed using a recursive space subdivision technique [16]. The user selects a bounding box within which to display the surface. In addition, the granularity of the PLA is controlled by selecting the maximum edge length of the approximating polygons, and the depth of the recursion.

- *Curves in Parametric Form*
  A rational plane curve given by parametric equations $x = \frac{f_1(s)}{f_3(s)}, y = \frac{f_2(s)}{f_3(s)}$ can be graphed over a parameter interval $[a,b] \subseteq R$. In Ganith, a PLA can be constructed by constant or adaptive stepping of the parameter in the given interval. The adaptive method produces smoother displays, because stepping is curvature dependent. Curve portions of higher curvature will produce denser subdivisions of corresponding portions of the parameter interval.

  Rational space curves given by a triple of rational functions of a single parameter can also be graphed in a similar way. The adaptive technique used is from [15]; see also [24],[28].

- *Surfaces in Parametric Form*
  A rational surface given by a parameterization

$$x = \frac{g_1(s,t)}{g_4(s,t)}, y = \frac{g_2(s,t)}{g_4(s,t)}, z = \frac{g_3(s,t)}{g_4(s,t)}$$

  can be graphed over a parameter rectangle $[a_1,b_1]\times[a_2,b_2] \subseteq R^2$. Once again, constant or adaptive subdivision methods may be chosen in constructing a PLA. The adaptive methods are curvature dependent, and are based on formulas given in [15]. These formulas are used by a stepping algorithm to find an appropriate set of points in the parameter rectangle. A Delaunay triangulation ([23]) of these points is constructed, and the triangles are mapped into space via the parameterization. This yields a curvature dependent triangulation of the surface.

2.2.2 Animation

Anything displayed by Ganith can be freely rotated by simple mouse motions. In addition, certain powerful animation tools are provided. These tools allow one to view entire families of curves and surfaces, thereby gaining geometric insight into a variety of situation.

- *Animation Objects and Groups*
  All displayable objects such as curves and surfaces can be converted by the user into *animation objects* by attaching an *animation script* (*script* for short). A script specifies a sequence of transformations to apply to the object. Once some animation objects are constructed, they can be collected into *animation groups* for simultaneous animation.

- *Animation Scripts*
  An animation script is a recipe for smoothly transforming the object in some way. Affine transformations (rotation, translation, scaling) are supported, as are a more interesting kind, coefficient variations. In the latter, some coefficient of the object is allowed vary in some range. This gives rise to a whole family of related curves or surfaces. For instance, one could visually examine the sensitivity of a singularity of a curve or surface to changes in a certain coefficient.

- *Intersection of Animation Objects*
  Given two animated objects, under certain conditions, one may intersect them. That is, at each time step, the corresponding "frames" of each animation object are intersected. For instance, one could attach scripts to two surfaces that translate them towards each other, and intersect the resulting animation objects. This would produce a sequence of intersection curves as the surfaces "pass through" one another. Much more complicated examples can be generated by combining the affine transformations with coefficient variation.

  All intersections allowed in Ganith may be applied to animated objects (currently, one is allowed to intersect two implicit plane curves or two implicit surfaces: see section 3).

- *Animation Movies*
  Since some animation operations cannot be done in real-time, a "movie" feature allows one to save the frames of an animation sequence. Once such a "movie" is made, it can be played forward or backward under user control, at real-time speed.

2.3 Symbolic Computation

Ganith provides several operations on curves and surfaces. The algorithms used are primarily algebraic in nature, but numerical calculations (especially root-finding) are crucial to their implementation.

2.3.1 Intersection by Birational Maps

Algorithms are provided for intersecting two curves or two surfaces, given in implicit form. These are based on the computation of birational maps by polynomial remainder sequences [4, 8].

Given two equations $f_1(x_1, \ldots, x_n) = f_2(x_1, \ldots, x_n) = 0$, the birational map computation reduces this system to an equivalent one of the form

$$\begin{aligned} Res(x_1, \ldots, x_{n-1}) &= 0 \\ x_n &= \frac{g(x_1, \ldots, x_{n-1})}{h(x_1, \ldots, x_{n-1})} \end{aligned}$$

Here $Res(x_1, \ldots, x_{n-1})$ is Sylvester's resultant of the original equations with respect to x_n. The resultant and the rational function g/h can be obtained by computing the subresultant polynomial remainder sequence [18],[19] of the polynomials f_1, f_2.

By this method, $n-1$ coordinates of the solutions of the system are given as solutions of a single equation. The remaining coordinate is given as a rational function of the others. Hence if one can "solve" a single equation in $n-1$ variables, one can solve two equations in n variables. This method can be generalized to more than two equations, at the cost of greater complexity [17],[11].

The curve and surface intersection facilities make use of this method. This method was chosen because of its efficiency and simplicity (only a single polynomial remainder sequence is needed).

- Curve Intersection
  Given two curves, the resultant and rational function are constructed. The resultant is a univariate polynomial. A numerical procedure is applied to find roots of this polynomial, which are then used to find common curve points. The two curves are displayed with their common points circled.

- Surface Intersection
  Given two surfaces, the resultant and rational function are constructed as for curves. This time the resultant is a plane curve. The numerical plane curve graphing procedure described in section 2 is applied, and the plane curve is mapped to a space curve by the rational function. The space curve is then displayed.

2.3.2 Global Parameterization

Given a curve or surface in implicit form, one can sometimes find a rational parameterization. Parameterization algorithms have been given in [1],[2],[3],[4],[29]. One algorithm for parameterizing quadratic hypersurfaces[9] has been implemented in Ganith. The user can immediately choose to display one or two dimensional sections of the parameterization using the facilities of section 2.

2.3.3 Local Parameterization at Singularities

Around any point of a plane curve, one can find power series that approximate the curve locally near that point [26, 21]. Near singular points, this is not easy; algorithms for this have been given in [7], [22],[20]. The algorithm of [7] has been implemented in Ganith. It uses fast algorithms for Weierstrass preparation, Hensel lifting, and Newton factorization to compute power-series approximations for each branch of the curve near a given singular point. The curve and the branches near the singular point are displayed.

2.3.4 Surface Fitting with Algebraic Surface Patches

C^1 Interpolation and Least-Squares approximation routines [13, 10, 14, 27] have been implemented in Ganith.

2.4 System Design

There are four major components of Ganith: user interface, controller, numerical and graphics subsystem, and algebra subsystem. The first three components reside in one process, and the algebra subsystem in another process. The algebra subsystem is written in Common Lisp, and the others in C and Fortran. All graphics and numerical calculations are performed in C or Fortran, while symbolic and exact calculations are done in Lisp. The two processes communicate with each other using Unix sockets, and may run on different hosts.

To start Ganith, a C program containing the user interface, controller, and numerical subsystem is executed. This invokes the algebra subsystem in a Lisp process, possibly on another host. The user interface is used for command input and output. Once a command is entered and edited to satisfaction, the controller sends it to the numerical or algebraic subsystems for execution, then the results are sent to the user interface for display. The algebraic and numerical subsystems may request computations from each other via the controller.

Now two aspects of Ganith are discussed, namely extensibility, and communication with other systems under development here.

2.5 Extensibility

In Ganith, our goal is not to provide a full-blown computer algebra program (e.g. MACSYMA, Maple, Mathematica etc). Instead Ganith, provides specialized capabilities that are useful in conjunction with other CA systems. For this reason, it was deemed unnecessary at this point to provide a new language for extending Ganith. Instead, some primitive

3 USER'S MANUAL

3.1 The User Interface

High level user interfaces are a developing trend in CA systems. In Ganith the user interface was a part of the initial design. It is implemented using the Athena Widget Toolkit for the X-11 window system, and a library that exploits special graphics hardware (such as polygon shading hardware) where available. See accompanying figure. The user interface is fully customizable to the extent allowed by the Widget Toolkit. For instance, the user can customize features of subwindows such as size, relative position, label, color, and keystroke interpretation.

The principal components of the user interface are discretization, visualization and input/output. Their capabilities are described below, and their usage is described in the Reference Manual section. The user interface also implements half of the protocol used in communicating with the Lisp process containing the Computer Algebra library.

3.1.1 Discretization

This component implements the display of graphical objects. Such objects are usually discretized approximations of piecewise-continuous mathematical entities. For instance, the set of real solutions of the equation $f(x, y) = 0$ describes a plane curve. This set may be unbounded, but the display area is finite. A portion of the set must be selected for graphing. A typical choice is the part lying inside a square centered around the origin. This portion is then approximated in some way, generally by a piecewise linear approximation.

The discretization component provides ways of generating such representations of mathematical entities. At present the following entities can be processed:

- Implicitly Defined Plane Curves
  The real solutions of $f(x, y) = 0$.

- Parametrically Defined Curves
  A plane curve may be defined as the set of points $\{[x(t), y(t)]\}$ where t, the *parameter*, varies over some interval. A space curve may be defined similarly as a set of points $\{[x(t), y(t), z(t)]\}$ in 3-space. The functions x, y, z are rational functions, that is, quotients of two polynomials in t.

- Implicitly Defined Surfaces
  The real solutions of $f(x, y, z) = 0$.

- Parametrically Defined Surfaces
  Like curves, surfaces may be defined parametrically as the set of points $\{[x(s,t), y(s,t), z(s,t)]\}$ where the parameters s and t each vary over some interval.

- Intersections of Implicitly Defined Plane Curves
  The real solutions of $\{f_1(x, y) = 0, f_2(x, y) = 0\}$. Here the plane curves of f_1 and f_2 will be graphed, and their common points circled.

- Intersections of Implicitly Defined Surfaces
  The real solutions of $\{f_1(x,y,z) = 0, f_2(x,y,z) = 0\}$. The intersection of two surfaces is in general a curve lying in 3-space.

Several variables control the concrete visual representation of these entities. These are bounding box corners, parameter ranges, parameter nstep sizes, stepping methods, subdivision levels, etc.

3.1.2 Visualization

Once a mathematical entity is approximated, the visualization component provides for its display and manipulation. The primary tools supported are *objects* and *windows*. An *object* is indirectly created by Ganith whenever some mathematical entity is approximated. A *window* is directly created by the user. Ganith maintains a list of objects and a list of windows, which are independently accessed and manipulated. Once a window is created, the user may *select* any number of objects into that window for display. Thus *selection* is the only operation that combines windows and objects.

Once an object is selected into a window, it can scaled and rotated along with all other objects in that window. Each window maintains its own scale factor and orientation vector. All objects of a window are drawn to the same scale and with in the same orientation.

An object consists of a set of equations, a set of *points* (i.e. an approximation to the entity described by the equations), and a *state* (i.e. the particular parameters used for generating the points). Objects may be *refined*: if the object's state variables are out of date, the current state is used to recompute the object's points. Objects may also be transferred to and from files.

3.1.3 Input/Output

Ganith departs from the line-oriented input style of traditional CA sytems. The Input window is an Emacs-like editor. It is mouse-sensitive, i.e. the mouse can be used for cut and paste operations. The keystroke translations of the editor are customizable.

The Output window is not editable. It is used to display numerical and symbolic results of computations, and to annotate visualizations displayed in the Graphics window.

Both Input and Output windows have a built-in history mechanism accessible via mouse scrollbars. The entire history of the Input window is editable. The history of both windows may be accessed in cut/paste operations.

File operations are available for inserting files into the Input window, and for saving the Output window contents.

3.2 The Computer Algebra Library

Most of Ganith's algorithms are implemented in Lisp, and form the Computer Algebra library. The library is structured into layers, in order of ascending algorithm complexity. Each layer uses and builds on the functionality provided by previous layers.

The library provides an abstract data type for polynomials. This data type is based on the recursive notation for polynomials: multivariate polynomials are represented as univariates whose coefficients are recursively polynomials of the same form in other variables, or constants.

All operations use this data type for polynomials. Accessor functions are used to query polynomials for characteristics such as degree and variable lists, while constructor functions are used to construct new polynomials. This structured approach localizes the exact details of the polynomial data structure in a few select functions. However, for the sake of efficiency, many low-level algorithms were constructed to be most efficient for sparse, recursively represented polynomials.

The layers of the library are now described in degree of ascending complexity. Algorithms marked with an asterisk are unimplemented at this time but shown to mark their place in the hierarchy. Some algorithms are separately implemented but not yet incorporated into Ganith.

- Polynomial Manipulation
  - Abstract data types
  - Accessor and constructor functions
  - Rational and modular constant arithmetic
  - Add, subtract, multiply, divide, pseudo-divide
  - Differentiate, evaluate, interpolate, Chinese remainder
  - Variable ordering, coefficient extraction
- Low-level Algebra
  - Root extraction for univariate polynomials
  - Subresultant remainder sequence calculation
  - Resultant calculation by modular and non-modular methods
  - Polynomial GCD
  - Rational parameterization of quadrics in any number of variables
  - Rational parameterization of singular cubic curves, and all cubic surfaces
  - Implicitization of parametrically given curves or surfaces
  - Weierstrass preparation
  - Newton factorization
  - Hensel's lemma
  - Pade approximation
  - Groebner Bases*

  - Macaulay's Resultant*
- Algebraic Geometry Implementation
  - Curve/curve intersection using resultants and birational maps for curves in implicit form
  - Surface/Surface intersection using resultants and birational maps for surfaces in implicit form
  - Intersections of hypersurfaces given in the same form, when implicitization or parameterization successfully converts one hypersurface to the other form*
  - Power-series parameterizations of curves at singularities
  - Three-surface intersection using remainder sequences and birational maps*

In addition to the Computer Algebra library, a small amount of Lisp code implements half of the protocol used to communicate with the user interface process.

3.3 REFERENCE MANUAL

Ganith is started by running the user interface program, simply called "ganith." The user interface will also start a special Lisp process that contains the CA library, and initialize communications with it. After that it will create a multi-panel window on the X display host.

3.3.1 Terminology

Some simple terminology is given for users unfamiliar with window systems. Only a small subset of window related operations are described, just enough to operate Ganith. Since such operations are customizable in X, these may vary from user to user, and we only describe some common defaults. Refer to X documentation for advanced use.

- *Mouse Clicking* This means depressing a mouse button and releasing it. The mouse is assumed to have three buttons, denoted here as LEFT, MIDDLE, and RIGHT. "Clicking" without any modifiers refers to clicking the LEFT mouse button. "Double-clicking" means clicking a button twice, rapidly, and "triple-clicking" is similar.
- *Text Selection* A contiguous portion of text may be selected by clicking the LEFT button at the beginning of the extent, and the RIGHT button at the end of the extent. A word can be selected by double-clicking on it, and a line by triple-clicking.
- *Cut and Paste* The current text selection may be cut (deleted) by typing "Control-W." It may be pasted (inserted) using "control-Y."

- *Buttons* Not to be confused with mouse buttons. These are control subwindows that perform some action in response to mouse clicks. Usually nothing else can be done in Ganith while a button is performing an action; while a button is active, its foreground and background colors will be reversed. There are several types of buttons:
  - *Command* This type will simply perform some action
  - *Selection* Selection buttons operate on the current text selection (see below)
  - *Dialog* These buttons will prompt the user for some input before performing an action
  - *Toggle* These buttons change some boolean state variable in Ganith. They have a dark background and light foreground when the state variable is true, and vice-versa when it is false.
  - *Menu* A menu button brings up a list of items for selection. There are two kinds of menus: zero-or-one, and zero-or-many. The first kind will only allow one item to be selected at a time. In each case, when an item is selected in the menu by clicking the LEFT button, it will be highlighted. Selecting an already highlighted item un-selects it. When all requisite items are selected, the "Ok" button must be pressed to continue the action. A "Cancel" button allows one to abort without performing the action. Finally, double-clicking an item in a zero-or-one menu is shorthand for selecting it *and* clicking the "Ok" button.

3.3.2 Organization

Ganith consists of (at least) three subwindows, and various buttons.

The major subwindows are the following:

Control Panel: Titled "Ganith Control Panel," this contains an assortment of buttons. This window may be resized to any shape and the buttons will attempt to arrange themselves to fit the shape.

Input/Output: This contains two text windows, with scroll bars. The upper text window is the **Input** window. Ganith commands may be entered here. It is a text editor that recognizes most of the standard editing commands of the EMACS editor. Click anywhere in the window to set the cursor location. Inserting and deleting characters from the keyboard is done relative to the cursor. Text may also be cut and pasted in this window. *Backspace* deletes the previous character. The lower text window is the **Output** window. Nothing should be typed here: Ganith uses it to print textual results. Both may be scrolled back and forth using the scroll bars (by holding down the MIDDLE mouse button in the bar and moving the mouse back and forth).

Graphics: These are titled "Ganith Window N" and are always square. There is always one window that is *current*. Graphical objects are always selected into the current window. Each window's title indicates whether it is current, and also contains a list of the objects selected by that window. The graphics windows may be resized, but will always be forced to remain square. Each window displays its own XYZ-space. The X axis runs horizontally,

Y vertically, and the Z-axis increases out of the screen. The initial viewpoint is usually at $(X = 0, Y = 0, Z = 20)$, looking at the origin.

Each window maintains its own scale and orientation, and all objects selected by that window are drawn accordingly inside it. An object may be selected by multiple windows and displayed in multiple sizes and orientations. A window's orientation may be changed by mouse motion inside the window (it doesn't have to be current), followed by clicking the LEFT button. If the user moves the mouse inside the window but doesn't want those motions to be later interpreted as rotations, clicking the MIDDLE button cancels any accumulated mouse movement. The user may also hold down the LEFT button and "drag" the mouse, causing continuous rotation of objects inside the window.

All rotations are performed around the axis in the XY-plane that is perpendicular to the direction of the mouse motion. In particular, vertical motion corresponds to rotation around the X axis, and horizontal motion corresponds to rotation around the Y-axis. A mouse motion whose length equals the length (or width) of the window corresponds to a rotation of 360 degrees.

The scale factor of the current window may be changed by the "Zoom" button, as described below.

Planar objects are also drawn in three-dimensional space, hence to view them without distortion, they must be viewed from along the Z axis. See the "Init" button below.

3.3.3 Interacting with Ganith

Ganith is operated using various control buttons. Each button is described along with its type (and default value, if it is a toggle).

- **Read**(Dialog) Insert a file into the Ganith input buffer.
- **Write**(Dialog) Write the contents of the output buffer into a file
- **Help**(Dialog)
- **Quit**(Dialog)
- **Execute**(Selection) Execute the text selection as a *command.* Might create a new object.
- **DrawCurve**(Selection) The text selected must be an expression describing a curve to be drawn. Creates a new object.
- **DrawSurface**(Selection) The text selected must be an expression describing a surface to be drawn. Creates a new object.
- **Init**(Command) Resets the viewpoint, scale, and orientation of the current window to its initial position. Thus to view a plane curve without distortion after rotating an object in three-dimensions, one clicks **Init** to reset the viewpoint to its initial position along the Z axis.

- **Clear**(Command) Clear the current window.
- **Redisplay**(Command) Redraw all objects in current window.
- **Zoom**(Dialog) Will prompt for a scale factor. This number is then multiplied into the scale factor of the current window. Typing a factor larger than 1 will cause an object to grow; typing a factor smaller than 1 will cause an object to shrink. A good way to perform repeated zooms is to enter a zoom factor of $1 + -0.1$, and click the "Ok" button on the Zoom prompt repeatedly.
- **Graphics**(Dialog) This button is used to set Ganith variables that control curve and surface display. Different algorithms are used for polygonalizing implicit and parametric surfaces. The variables must be entered separated by commas. For each method the names, types, and default values of the variables are listed.
  - **Implicit Surfaces** An Octree decomposition method is used. This method starts with an initial bounding cube, and subdivides it recursively until small facets of the surface are found. The variables are
    - Coordinates of one cube corner: float : -2,-2,-2
    - Cube side length: float : 4
    - Maximum polygon side length: 0.5
    - Should normals be generated: 0 (no) 1 (yes) : 1
    - Minimum subdivision level : integer : 3
    - Maximum subdivision level : integer : 8

    Thus the default variable string is "$-2.0, -2.0, -2.0, 4.0, 0.5, 0, 3, 8.$"
  - **Parametric Surfaces** Either constant or adaptive stepping of the parameters may be used. The variables are
    - initial-S,final-S,initial-T,final-T: float : -1,1,-1,1
    - number of points in each parameter range : integer : 20
    - adaptive delta : float : 0.3

    If adaptive delta is 0.0, then constant stepping with the given number of points will be used. Otherwise the points value will be ignored, and stepping will proceed by increments of delta, scaled by the curvature of the surface. The default variable string is "$-1.0, 1.0, -1.0, 1.0, 20, 0.3$" (adaptive stepping is the default).
  - **Parametric Curves** If a curve is being drawn, all values must be given, but only "initial-S," "final-S" and "points" are used.
- **Colors**(Command) Ganith maintains a small, fixed set of colors, and this button allows the user to select the current line drawing color from a menu.

- **Shades**(Command) Ganith maintains a small, fixed set of color shades, and this button allows the user to select the current polygon shade from a menu.

The following buttons are toggles.

- **AutoRedraw**(Toggle,True) When true, redraw the screen each time some value such as zoom factor or viewpoint changes.
- **AutoImp**(Toggle,True) When true, expressions given to "DrawCurve" and "DrawSurface" must be implicit definitions of curves and surfaces. When false, they must be expressions for parametrics.
- **AutoCull**(Toggle,False) When true, polygons of a displayed surface whose normals point away from the viewing direction will be removed.
- **AutoShade**(Toggle,False) When true, polygons of a displayed surface are filled according to the current shading parameters.
- **AutoWire**(Toggle,True) When true, edges of polygons of a displayed surface are drawn.
- **AutoSelect**(Toggle,True) When true, every object created is automatically selected into the current window.

The following buttons relate to windows and objects.

- **SelectWindow**(Menu,0/1) The window list is displayed for selection, along with a special item called **NEW**. One item is selected to be the current window. If **NEW** is selected, a new window is created and made current.
- **DeleteWindows**(Menu,$0/\infty$) The window list is displayed, and several windows may be selected for destruction. One may not destroy all windows – at least one must be left. If the current window is destroyed, some other window is made current.
- **SelectObjects**(Menu,$0/\infty$) The object list is displayed, and several objects may be selected for inclusion into the current window.
- **UnSelectObjects**(Menu,$0/\infty$) The object list of the current window is displayed, and several objects may removed from the current window.
- **DeleteObjects**(Menu,$0/\infty$) The object list is displayed, and several objects may be selected for destruction. Destroyed objects are removed from each window that had selected them.
- **RefineObjects**(Menu,$0/\infty$) The object list is displayed, and several objects may be selected for refinement. Each object is refined and then redrawn in each window that contained it. For successful refinement, an object must contain both the equations part and the state part. The state part is compared to the current graphics state and the points of the object are revised if the object's state is out of date.

- **DescribeObjects**(Menu,0/∞) The object list is displayed, and several objects may be selected. For each object, its equations, points, and state (whichever are present) are described.

- **InvertObjects**(Menu,0/∞) The object list is displayed, and several objects may be selected. For each object, the direction of all polygon normals (if any) are reversed.

- **RenameObject**(Menu,0/1) The object list is displayed, and one object may be selected for renaming. After selection, a dialog is displayed into which a new name may be entered. The name must not already be in use. Objects are given numeric names by default.

- **SaveObject**(Menu,0/1) The object list is displayed, and one object can be selected for saving to disk, for subsequent retrieval and use. After selection, a dialog is displayed into which a file name prefix must be entered. Then, three files are created with the suffixes ".eq", ".pts", and ".st", containing the equations, points, and state of an object, respectively. If an object is missing one of these parts, the corresponding file is not created. The saved file format will be described in the programmer's guide.

- **ReadObject**(Dialog) A dialog is displayed into which the user enters a file name prefix. Ganith searches for files with that prefix and suffixes ".eq", ".pts", and ".st". All three need not be present, but at least one of ".eq" and ".pts" must be present. If no syntax errors are detected in these files, a new object is created.

3.3.4 Commands and Expressions

- **Polynomials** Ganith understands limited macsyma-like expressions. Floating-point and rational coefficients are allowed. However, only polynomials are recognized, hence "`1/2*(x-0.3*y)*z`" is a valid expression but "`x/2`" is not.

- **Variables** The "space" where curves and surfaces are manipulated has its axes labeled, by default, X, Y, and Z. The default parameter space is S,T. All expressions describing plane curves must be in X and Y, when implicitly defined, or in S, when parametrically defined. Likewise, polynomials describing surfaces must be in X,Y, and Z or in S and T.

  These variable bindings can be changed, as described later. All variables are translated into upper-case, hence their case does not matter (but command names must be in lower-case).

- **Expressions** An expression for an implicitly defined curve is a polynomial in two variables; for a surface, a polynomial in three. Not all variables must be present (then the object will be a cylinder). A parametrically defined curve or surface is represented by a tuple $[r_1, r_2]$ or $[r_1, r_2, r_3]$ respectively. Each r_i is an expression describing a rational function; it can be simply a polynomial, or of the form "polynomial1—polynomial2," where "polynomial1" is the numerator, and "polynomial2" is the denominator.

- **Commands** The text of these must be selected, and executed using the **Execute** button. In what is below, p_i stands for a polynomial. Only implicit-implicit intersections are handled. The commands currently supported are:
  - **intersect2e2d**(p_1,p_2) Intersect two plane algebraic curves and display their points of intersection, along with the curves. This will create an object which will be refined if the plane curve bounding box is changed.
  - **intersect2e3d**(p_1,p_2) Intersect the surfaces p_1 and p_2, displaying their curve of intersection. The intersection curve is projected onto a plane curve first, so this object can be refined by changing the plane curve bounding box.
  - **intersect3e3d**(p_1,p_2,p_3) Intersect the surfaces p_1, p_2, and p_3. Currently, this is just a shortcut for displaying their pairwise intersections (i.e. calling **intersect2e3d** three times).
  - **param2dNv**($poly, v_1, \ldots, v_{N-1}, p_1, \ldots, p_N$) Parameterize the hypersurface of degree 2 and dimension N defined by the equation $poly = 0$. The v_i are the parameter variables to be used, and the p_i are coordinates of a point on the hypersurface. The parameterization is displayed in the (textual) output window.
  - **compactify**(p) Compactify the polynomial p which may be in two or three variables. If it is in two variables x, y, then it is homogenized with the variable z, and the intersection of this surface with $x^2 + y^2 + z^2 - 1$ is displayed. Otherwise a homogenizing variable w is used and the resultant of the homogeneous surface and $x^2 + y^2 + z^2 + w^2 - 1$ with respect to w, is displayed.
  - **eliminate**(p_1, p_2, v) Eliminate the variable v from the equations $p_1 = 0$ and $p_2 = 0$, by Sylvester's resultant. The eliminant is displayed in the (textual) output window.
  - **realify2d**(p) Compute and return the real and imaginary parts of a curve, and their resultant. If p is a polynomial in two variables x and y, the substitution

$$\begin{aligned} x &= x + iw \\ y &= y + iz \end{aligned}$$

    is made; then the real and imaginary parts are computed, and their resultant w.r.t. w is returned. Their resultant is a polynomial in three variables defining a surface, which is displayed.
  - **curvewin**($\min_x, \min_y, \max_x, \max_y$) This resets the bounding box for implicit plane curve display.
  - **setvars**(v_1, v_2, v_3) The names of the axes of the plane are set to (v_1, v_2) and those of space are set to (v_1, v_2, v_3). Implicit curves and surfaces must use these variables. The variables are (x, y, z) by default.
  - **setpvars**(v_1, v_2) The variables for the parametric spaces of dimensions 1 and 2 are set to v_1 and (v_1, v_2) respectively. The variables are (s, t) by default.

Some simple examples of commands and expressions are listed below.

- **intersect2e2d**(x^2+y^2-1,y^3-x)
- **intersect2e3d**(x^2+y^2+z^2-1,z^2-y^3)
- **intersect3e3d**(z^2-y^3,x^2+y^2+z^2-1,(x-1)^2+y^2+z^2-1)
- **param2dNv**(x^2+y^2-1,s,1,0)
- **setvars**(u,v,w)
- **setpvars**(u,v)

3.4 Enhancements

Ganith graphics is built on top of a library that provides a device-independent graphics interface. This library is currently implemented for the SGI 4D class workstations. In that version of Ganith, additional features provided are hidden-surface elimination using Gouraud shading, and lighting models.

3.5 TUTORIAL

We present here a sample interaction with Ganith, in step-by-step fashion. The sequence of operations presented here are enough to gain some understanding of the Ganith user interface and some of Ganith's main facilities. The steps here are of course only a suggestion, and the user is encouraged to experiment with the various facilities described.

An interaction is now described in which the user draws two surfaces and intersects them, displaying these objects in multiple windows.

1. Start Ganith
2. Resize window 0 to make it a little smaller
3. Select a color, e.g. magenta by clicking the **Colors** button repeatedly.
4. Enter the text "1-x^2+y^2+z^4" in the input window, select it with the mouse, and click the **DrawSurface** button. A surface will be drawn in window 0, and object 0 will be created. Rename object 0 to "surf1" using the **Rename Object** button. A snapshot of the screen at this stage is shown in Figure 4.
5. Use the **Select Window** button to create a new window, called window 1.
6. Resize window 1 to make it smaller.
7. Select a different color, e.g. brown.

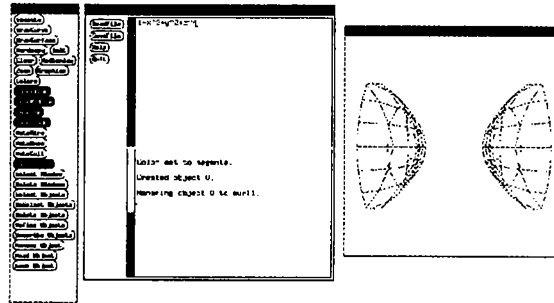

Figure 4: Output of a Draw Surface($1 - x^2 + y^2 + z^4$) Command in Window 0

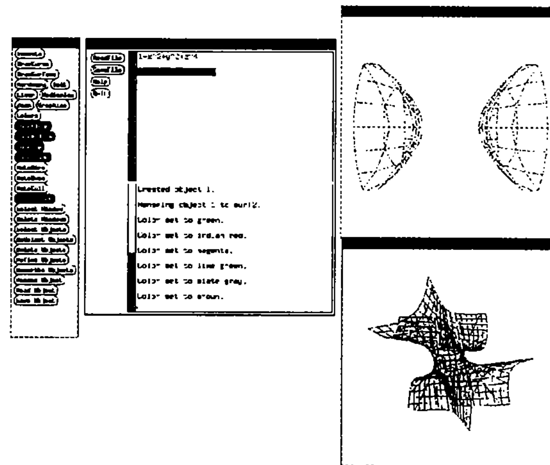

Figure 5: Draw Surface($x^2 + y^2 - 2 * x * y * z - 1$) Displayed in new Window 1

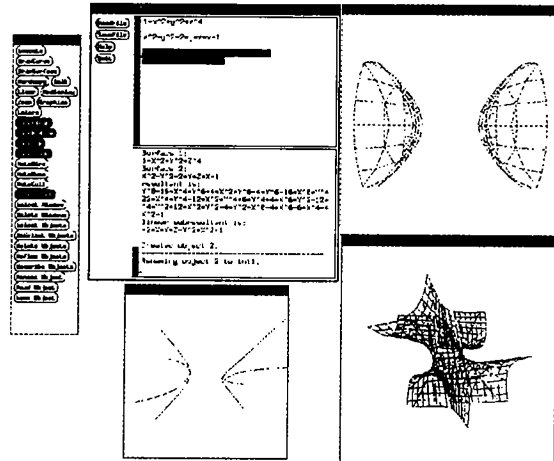

Figure 6: Intersect2e2d$(1 - x^2 + y^2 + z^4, x^2 + y^2 - 2 * x * y * z - 1)$ Displayed in Window 2

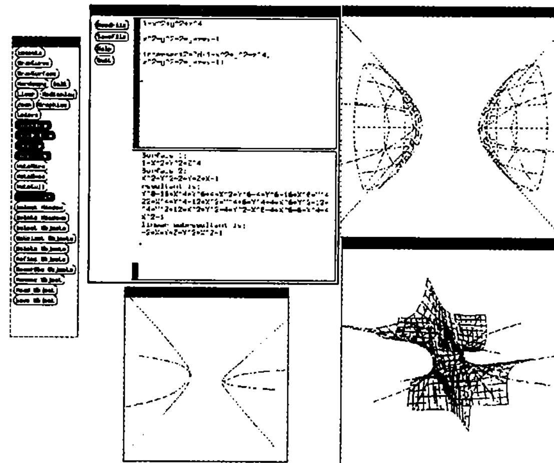

Figure 7: Simultaneous Display of Surface and Intersection Curve in Windows 0, 1

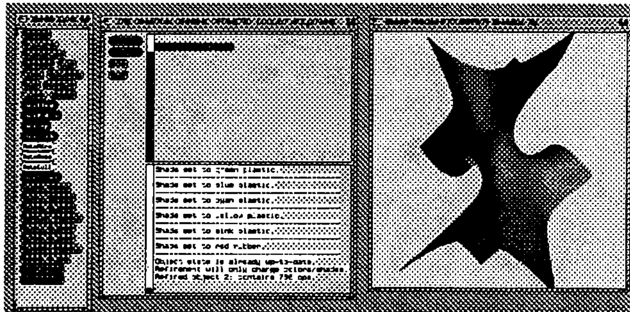

Figure 8: Shaded Display of Draw Surface($x^2 + y^2 - 2 * x * y * z - 1$)

8. Enter the text "`x^2+y^2-2*x*y*z-1`" into the input window and click the **DrawSurface** button. This will draw a surface into window 1, calling it object 1. Rename this object to "surf2."

9. Click the **Graphics** button and change the surface bounding box to have corner $-3, -3, -3$ and side of length 6, and refine the object "surf2" using the **RefineObject** button. A larger portion of the surface will now be shown in window 1. A snapshot of the screen at this stage is shown in Figure 5. On graphics workstations equipped with a Z-buffer, a shaded display of the same surface is possible and shown in Figure 8.

10. Create a new window, window 2, and resize it to be somewhat smaller.

11. Select a new color, e.g. blue.

12. Edit the contents of the input window to contain the text "`intersect2e2d(1-x^2+y^2+z^4,x^2+y^2-2*x*y*z-1)`." Then select this text with the mouse and click the **Execute** button. This will display the intersection curve of the surfaces in window 2, and create a corresponding object called "object 2." Rename the object to "int1." A snapshot of the screen at this stage is given in Figure 6.

13. Select window 0.

14. Select the object "int1" into window 0.

15. Select window 1.

16. Select the object "int1" into window 1. At this point the screen will show the curve of intersection by itself and also overlaid on each of the intersecting surfaces. If desired, one can change the drawing parameters and refine the object "int1" so that a larger portion of the intersection curve is displayed, for emphasis. A snapshot of the screen is shown in Figure 7.

4 PROGRAMMER'S GUIDE

Ganith is divided into two components, the user interface and the CA library. The components are written in separate languages and run inside separate processes. It is therefore logical to discuss them separately. However, we first discuss the design of Ganith.

The first implementation was on a Sun workstation running the Unix operating system. Later it was ported to the SGI 4D class of workstations. Ganith is written in Common Lisp and C. It uses the X11 window system via the Athena Widget toolkit. The Lisp part is written entirely in standard Common Lisp. In our experience, to deviate by even a minute amount from standard Common Lisp (by using implementation-dependent extensions) is to drastically reduce portability. Thus, while a certain amount of efficiency is lost, the system as a whole is quite portable. Ganith is meant to be portable to any system that supports Common Lisp and the X window system.

The Lisp process and the C process communicate via UNIX pipes and need not reside on the same host.

The user interface and CA library were separated for practical reasons. This separation allows graphics operations to be written in C, taking advantage of the extensive C libraries available for graphics on contemporary workstations. Likewise, Lisp is well suited for algebraic computation. Its advantages such as built-in symbol and large-number manipulation lead to quick prototyping and higher programmer productivity than a language such as C. However, the prices paid for using Lisp are high memory overhead, slower run-time execution, somewhat lower portability, and the cost of an extra process. At this time we are judging the cost of continuing to use Lisp versus writing a CA kernel from scratch in C.

The two components of Ganith work in a master-slave relationship. The user interface is the master; it accepts commands from the user and executes them. At times it may need facilities provided by the CA library. In this case, a simulated foreign-function call is executed to call a function in the Lisp process and access its return value. The simulated foreign-function call actually consists of two steps: send and receive. The send step returns immediately after sending the command to Lisp, without waiting for its completion. The receive call blocks until Lisp finishes its computation. This asynchrony may have some value on multiprocessor architectures.

The rest of this section is as follows. First the user interface and CA libraries are described, along with their directory structures. Next, the important functions of the user interface and CA library are listed. Then we describe how to add new functions to Ganith, and finally it is shown how to make the entire system.

4.1 User Interface Implementation and File Structure

The user interface is an X client. It is written using the Athena Widget Toolkit. The general style of interaction is similar to other Toolkit programs, which is to edit commands in some text window, and then perform some mouse action, such as clicking a button widget. Thus there is no explicit read-execute-print loop present in the user interface source code. The various actions are localized in individual *callback* procedures associated with each button

described in the user's guide.

Now we describe the files in the user interface.

- ui.c
  The user interface control structure, including the main procedure, is in this file. The main procedure makes itself an X client and initializes the Toolkit. It creates all the Ganith widgets, and attaches Callback procedures to each widget created. It also creates the Lisp subprocess. After initialization, control passes to a subroutine that multiplexes between several event sources, such as the X server, other Shastra processes, and other local graphics servers. The only time any user interface code is executed is when the user performs some action in a widget that has a callback procedure attached to it.

- actions.c
  Each command recognized by the **Execute** button of the user interface has a C function, called an *action procedure*, bound to it. A table that maps command names to action procedures is maintained by Ganith. This file contains code to implement the action procedure table lookup and utility functions to help in argument parsing.

- cmds1.c, cmds2.c
  Both these files contain various action procedures.

- draw.c
  A high-level device-independent graphics library is implemented in this file. This library supports *graphics objects* which are aggregates of graphics primitives such as point, line segment, polygon, etc.

- li.c
  This file contains the C/Lisp inter-process communication procedures. There are procedures to fork a new Lisp process, send data to the Lisp process, and receive data from the Lisp process. All data communicated in either direction are formulated into strings whose format obey a certain protocol. This protocol is implemented in li.c, and its Lisp counterpart in the the CA library code. The protocol simulates a foreign-function call.

- hl_graphics.c
  This file contains code to draw implicitly defined plane curves and space curves defined by an implicit plane curve and a birational map.

- lists.c
  This file contains code to maintain a global list of objects, a global list of windows, and all the operations supported on these items.

- io.c
  This file contains code to transfer objects between the user interface and the disk.

In addition to the above files, there are various subdirectories that each implement a certain module. We describe each module here. Each directory contains files used to build a library with a name of the form "lib*XXX*.a". Each of these libraries is linked into Ganith.

- Parametric/
  This directory contains files used to build a C library called "libpar.a." It implements drawing routines for parametrically defined curves and surfaces, using constant or adaptive (curvature-dependent) stepping.

- Octree/
  This directory contains files used to build a C library called "liboct.a." It contains routines for faceting an implicitly defined surface into polygons. The method used is a recursive subdivision of space, called Octree Decomposition.

- Polymath/
  This directory contains files used to build a C library called "libpoly.a." It contains a set of simple polynomial arithmetic routines written in C. These routines include parse, unparse, add, subtract, multiply, differentiate, and evaluate. They are used throughout the user interface.

- Roots/
  This directory contains files used to build a C library called "libroots.a." It contains a polynomial real zero finder.

- Menu/
  All the code to maintain and manipulate 0/1 and $0/\infty$ menus, as described in the user's guide, are contained here. A C library called "libmenu.a" is built.

- Network/
  The files here are used to build a C library called "libnet.a." This library implements communication with other Shastra programs, and the multiplexer subroutine invoked by the user interface.

- Shilp/
  Consists of include files representing data structures of Objects communicated over the network by the SHILP-Shastra program.

- Interpolate/
  The files here pertain to the functions responsible for C^1 interpolation of points, curves with normals using algebraic surfaces.

- Queen/
  The files here pertain to the functions responsible for C^1 smoothing interpolation of polyhedra using quintic algebraic triangular patches.

4.2 Computer Algebra Library Implementation

The CA library is implemented in a layered fashion. Procedures at a given level can use procedures defined in any lower level. At the heart of the library is multivariate polynomial manipulation.

All polynomials are represented in *Recursive Canonical Form* (*RCF*). In this form, a polynomial in the variables $x_1, \ldots, x_n$ is represented either as a constant, or as a polynomial in x_n whose coefficients are (recursively) polynomials in the remaining variables $x_1, \ldots, x_{n-1}$. The variable x_n is sometimes referred to as the *main variable.* A strength of this form (for purposes of implementation) is that multivariates "look like" univariates, making it easy to modify algorithms for univariate polynomials to handle multivariates.

The data structure used to represent RCF polynomials can be described as a variant record in a PASCAL-like language. We assume the types *number* and *symbol* are predefined with the obvious meanings. The keyword **listof** followed by a type name t denotes a composite type that is a sequence of items of type t.

```
type rcf =
    record
        case constantp : boolean of
        true : (constant : number);
        false : (nonconstant : record
                                    v : symbol;         (* main variable *)
                                    tl : listof term; (* term list *)
                                end);
    end;
type term =
    record
        c : rcf;        (* term coefficient *)
        e : integer;  (* term exponent *)
    end;
```

Now for some examples of this representation. Let items in a record be denoted by enclosing them in "[" and "]," and let items in a list be denoted by enclosing them in "(" and ")." Then we have the following correspondences between polynomials and their RCF representations:

$3 \quad \mapsto \quad 3$
$2x + 1 \quad \mapsto \quad [x, ([2,1], [1,0])]$
$-x^2 - 2y^2 - 3xy - 4 \quad \mapsto$
$[y, ([-2,2], [[x, ([1,-3])], 1], [[x, ([2,-1], [0,-4])], 0])]$

RCF polynomials don't have unique representations. For instance x^2 could be $[x, ([1,2])]$ or $[y, ([[x, ([1,2])], 0])]$. In the latter form x^2 is treated as a constant polynomial in y. Each low-level procedure that takes multiple RCF polynomials arguments will operate correctly only if the polynomials all contain the same variables, in the same order, i.e. they must

be from some the same polynomial domain $D[x_1, \ldots, x_n]$. Polynomials from the same domain may be referred to as *compatible* polynomials. Higher-level procedures must make polynomials conform to the appropriate variable set and ordering before using the low-level routines. This approach is used for speed. Variable reordering is not a common operation for "most" applications, hence it is made more expensive so more common operations can be performed faster.

Constant coefficients may be rationals or numbers modulo a prime. In the latter case constant arithmetic is performed in the appropriate finite field. This set of constants allows implementations of modular algorithms to use the same functions for polynomial arithmetic as non-modular algorithms.

We now describe the various layers and their files. Each layer, or level, has one directory containing all its files. The layer "level1" is at present undefined.

- level2/
  This level contains the "Polynomial Manipulation" layer. Polynomials are represented in RCF form. The files are
  - rcf_arith.lsp
    This file contains the data structure for RCF polynomials and basic arithmetic routines (add, subtract, multiply, divide, pseudo-divide).
  - rcf_math.lsp
    More operations for polynomials in RCF form: evaluate, interpolate, differentiate, Chinese remainder.
  - rcf_io.lsp
    Parsing and printing of RCF polynomials.
  - rcf_poly.lsp
    Non-arithmetical manipulation of RCF polynomials, such as coefficient extraction and variable ordering.
  - misc.lsp
    Some routines that don't fit anywhere else.
- level3/
  This directory contains the "Low-Level Algebra" layer. The files are
  - resultant.lsp
    Modular and non-modular methods for Sylvester's resultant; subresultant remainder sequence calculation, univariate GCD.
  - gcd.lsp
    Multivariate GCD.
  - param.lsp
    Rational parameterization of conics and higher dimensional hypersurfaces of degree 2.

- wnh.lsp
  Weierstrass Preparation, Newton Factorization, Hensel's Lemma. This code has been tested under Symbolics Common Lisp and has not yet been incorporated under Ganith.

- level4/
  This directory contains the "Algebraic Geometry" layer. The files are

  - solve.lsp
    This file contains routines to solve systems of two equations in two or three unknowns, using subresultant remainder sequences and birational maps.
  - intersect.lsp
    Routines that use solve.lsp (possibly repeatedly) to compute curve/curve and surface/surface intersections.
  - output.lsp
    This file implements the Lisp half of the inter-process protocol used in communicating with the user interface.
  - ci.lsp
    This file contains a stub for each exported function in the CA library, that is accessible to C via the simulated foreign-function call.

- environ/
  This directory contains the files for compiling lisp, and is the place where the loaded lisp binary is stored. The files in this directory are

  - make.lsp
    Compilation instructions.
  - env.lsp
    Environmental features, such as memory allocation.
  - init_ganith.lsp
    A load file.

4.3 C Side Functions

We now list the important functions from the user interface, listed by the file or library it belongs to. Within each listing, functions are grouped by functionality.

- ui.c
  The following functions are X callbacks, and take the standard arguments. They correspond to identically named buttons of the user interface.

  AbortCallback(w_button, client_data, call_data)
  AutoAxesCallback(w_button, client_data, call_data)

AutoClearCallback(w_button, client_data, call_data)
AutoCullCallback(w_button, client_data, call_data)
AutoImpCallback(w_button, client_data, call_data)
AutoRedrawCallback(w_button, client_data, call_data)
AutoShadeCallback(w_button, client_data, call_data)
AutoWireCallback(w_button, client_data, call_data)
ClearCallback(w_button, client_data, call_data)
ColorsCallback(w_button, client_data, call_data)
DialogCallback(w_button, client_data, call_data)
DrawCurveCallback(w_button, client_data, call_data)
DrawSurfaceCallback(w_button, client_data, call_data)
ExecuteCallback(w_button, client_data, call_data)
GraphicsCallback(w_button, client_data, call_data)
HardcopyCallback(w_button, client_data, call_data)
HelpCallback(w_button, client_data, call_data)
InitCallback(w_button, client_data, call_data)
QuitCallback(w_button, client_data, call_data)
ReadCallback(w_button, client_data, call_data)
RedisplayCallback(w_button, client_data, call_data)
SaveCallback(w_button, client_data, call_data)
ZoomCallback(w_button, client_data, call_data)

The following functions are used for printing in the output buffer.

AppendOutput(s)
AppendOutputLine(s)
BeginConversation()

Functions for drawing something into the current window.

DrawAxes()
DrawImplicitSurface(s)
DrawParametricCurve(s)
DrawParametricSurface(s)

Functions for manipulating various state variables kept by ganith.

LoadGrState(pgs)
SaveGrState(pgs)
SetVars(xyz)
SetPVars(st)

Some action procedures related to user interface state variables.

CurveWinUI(args)
SetPVarsUI(args)
SetVarsUI(args)

Utility functions.

NewString(s)
StrUpCase(str)
sopen
sgets

- actions.c
  These functions handle action procedure lookup and command argument parsing.

  GanithCommandHandler(cmd)
  GanithGetArgs(r, n, st)
  FreeArgs(argc, argv)

- cmds1.c, cmds2.c
  Action procedures from cmds1.c, and a corresponding procedure that actually implements the functionality, without the wrapper for making a graphical object, etc.

  Intersect2e2dUI(args)
  Intersect2e2d(c1, c2)
  Intersect2e3dUI(args)
  Intersect2e3d(s1, s2)
  Intersect3e3dUI(args)
  Intersect3e3d(s1, s2, s3)

  Action procedures from cmds2.c.

  CompactifyUI(args)
  Compactify(s, drew)
  EliminateUI(args)
  Param2dNvUI(args)
  Realify2dUI(args)
  Realify2d(s)

- draw.c
  Functions for creation and display of graphical objects (collections of graphics primitives).

  StartGrObjectDescription()
  EndGrObjectDescription()
  DrawGrObject(pgo, orient, scale)
  FreeGrObject(pgo)

  Functions for drawing graphics primitives and optionally saving them into graphical objects.

  PointAbs(x, y, z)
  LineAbs(x0, y0, z0, x1, y1, z1)
  LabelledPointAbs(x, y, z)
  DrawPolygon(num_vertex, vertices, normals)
  DrawColor(rgb)

  Other functions for changing graphical objects.

  RecolorGrObject(pgo)
  InvertGrObject(pgo)

- li.c
  Functions for starting up lisp, calling lisp, and receiving data returned from lisp.

  InitializeLispInterface(host)
  CallLispFunction(fn, argc, argv)
  GetLispReturn(out_buf)

- hl_graphics.c
  Functions for drawing implicitly defined plane and space curves, and setting curve drawing parameters.

  SetPlaneCurveParams(x_min, x_max, y_min, y_max)
  DrawImplicitPlaneCurve(curve)
  DrawImplicitSpaceCurve(pcurve, cs1, cs0, s1, s2, a, b, e, e2)

- lists.c
  Callbacks for the buttons from user interface.

AutoSelectCallback(w_button, client_data, call_data)
DeleteObjectsCallback(w_button, client_data, call_data)
DeleteWindowsCallback(w_button, client_data, call_data)
DescribeObjectsCallback(w_button, client_data, call_data)
InvertObjectsCallback(w_button, client_data, call_data)
RefineObjectsCallback(w_button, client_data, call_data)
RenameObjectCallback(w_button, client_data, call_data)
SaveObjectCallback(w_button, client_data, call_data)
SelectObjectsCallback(w_button, client_data, call_data)
SelectWindowCallback(w_button, client_data, call_data)
UnSelectObjectsCallback(w_button, client_data, call_data)

Functions for manipulating windows and objects.

UIAddWindow()
UIAddObject(pgo, gt, neqns, eqns, select)
UIDeleteObject(oid)
UIDeleteWindow(wid)
UIDescribeObject(oid)
UIInvertObject(oid)
UIMakeWinCurrent(wid)
UIReadObject(file_prefix)
UIRedrawWindow(wid)
UIRedrawWindows(wids, n)
UIRefineObject(oid)
UIRenameObject(oid, new_name)
UIResetWindow(wid)
UIRetitleWindow(wid, current)
UIRotateWindow(wid, angle, axes)
UISaveObject(oid, file_prefix)
UIScaleWindow(wid, scale)
UISelectObject(oid, wid)
UISelectWindow(wid)
UIUnSelectObject(oid, wid)
UIWinFromXS(xs_id, pwid)

- io.c

  Functions for reading and writing the three parts of objects to and from files.

  ReadObjectEqns(pnvars, pvars, pneqns, peqns, fp)
  ReadObjectPts(ppgo, fp)
  ReadObjectSt(pgt, ppgs, fp)

ReadStringFile(fp, term)
SaveObjectEqns(po, fp)
SaveObjectPts(po, fp)
SaveObjectSt(po, fp)

- libpar.a
  Functions for drawing parametric curves and surfaces.

  DrawParamCurve(curve,is,fs,pp,delphi,min_step,max_step)
  DrawParamSurface(surface,is,fs,it,ft,pp,delphi,min_step,max_step)

- liboct.a
  A function for octree decomposition of an implicit surface.

  OCtree(octree_info, equation)

- libpoly.a
  Polynomial arithmetic and manipulation.

  AddPoly(x,y)
  ConformPoly(p1, p2)
  ConformPolyToVars(vars, varnames, p)
  CopyPoly (polyc)
  CreateConstantPoly(vars, varnames, coeff)
  CreateMonPoly(varname)
  DestrPoly(deadpoly,terms)
  DiffPoly(x,var)
  EvalPoly(x,point)
  ExpPoly(p, e)
  MultPoly(x,y)
  NegPoly(pospoly)
  Parse(s)
  Partials(x)
  SubPoly(x,y)
  UnParse(x)

- libroots.a
  Polynomial real root finding.

  CRealRoots(poly_s)

- libmenu.a
  Functions for creating and manipulating lists of strings, and popping them up as 0/1 or $0/\infty$ menus.

  MLAddEntries(mid,newlist,num)
  MLAddMenu(pmid)
  MLChooseMany(mid, func)
  MLChooseOne(mid, func)
  MLDeleteEntries(mid, list, num)
  MLDestroyMenu(mid)
  MLFreeMenu(Menu,Size)
  MLInit(xac, wgParent)
  MLRetrieveMenu(mid, list, psize)

- libnet.a
  Functions for making connections to other Shastra processes and multiplexing among them.

  mplexInit()
  mplexRegisterChannel(fd,handler,arg)
  mplexUnRegisterChannel(fd)
  mplexMain (flushFunc)
  mplexGetFilePtrs(fd,pInStream,pOutStream)

4.4 Lisp Side Functions

Now the important functions from the CA library are listed by file, and again grouped by functionality.

- level2/

  - rcf_arith.lsp
    Functions for coefficient arithmetic, over the field of rational numbers or a fixed finite field.

    rcf-set-coefficient-modulus (p)
    coeff+ (a b)
    coeff- (a b)
    coeff* (a b)
    coeff/ (a b)
    coeff-negate (a)
    coeff^(a e)

Predicates on polynomials.

rcf-constant (poly)
rcf-zerop (poly)

Constructor and accessor functions for polynomial data structures (terms, term lists, and polynomials).

make-term (e c)
term-e (term)
term-c (term)
make-terms (n)
make-tl (n terms)
tl-length (tl)
tl-terms (tl)
terms-ref (terms index)
make-rcf-nc (v tl)
rcf-v (poly)
rcf-tl (poly)
rcf-ldcf (poly)
rcf-degree (poly)
rcf-degree-v (poly v)

Functions for arithmetic on RCF polynomials.

rcf-add (poly1 poly2)
rcf-subtract (poly1 poly2)
rcf-negate (poly)
rcf-multiply (poly1 poly2)
rcf-exponentiate (poly n)
rcf-divide (poly1 poly2)
rcf-pseudo-divide (poly1 poly2)
rcf-remainder (poly1 poly2)
rcf-quotient (poly1 poly2)
rcf-pseudo-quotient (poly1 poly2)
rcf-pseudo-remainder (poly1 poly2)

A few other polynomial operations.

rcf-reduce-coefficients (poly)
rcf-norm (poly)
rcf-max-norm (poly)

- rcf_math.lsp
  Higher level math routines.

rcf-evaluate (poly v val)
rcf-substitute (poly s_list)
rcf-partial-differentiate (poly v)
rcf-interpolate (points vals var n)
rcf-make-linear-poly (var c)
rcf-chinese-remainder-constant-moduli (residues moduli n)
rcf-primitive-part-1 (poly)
rcf-content-1 (poly)
rcf-realroots-of-univariate (inpoly)

- rcf_io.lsp
  Functions for lexical analysis, parsing and printing of polynomials in RCF form, from and to a string.

  lex (str)
  rcf-parse (str)
  rcf-format (poly)

- rcf_poly.lsp
  Functions for symbolic operations on polynomials.

  rcf-make-monomial (var e c)
  rcf-atom-to-poly (a)
  rcf-var-order-< (v1 v2)
  rcf-var-order-> (v1 v2)
  rcf-var-order-= (v1 v2)
  rcf-vars (rcf)
  rcf-reorder-mainvar (poly)
  rcf-constant-term (poly)
  rcf-coeff (poly var e)

- misc.lsp
  Some utility functions.

  make-simple-string (s)
  make-adjustable-string ()
  reset-random-state ()

- level3/

  - resultant.lsp
    Functions for modular and non-modular calculation of resultants, subresultant chains, etc.

subresultant-coefficient-bound (poly1 poly2 j)
subresultant-modular (poly1 poly2 j)
subresultant-prs (poly1 poly2)
multivariate-resultant (poly1 poly2 j)
resultant (poly1 poly2 j var)

- gcd.lsp
  Multivariate gcd computation and related functions.

  rcf-gcd (poly1 poly2)
  rcf-primitive-part (poly)
  rcf-content (poly)

- param.lsp
  Functions for parameterizing implicit curves and surfaces of low degree.

  parameterize (poly vl pvl optionals)
  parameterize-trivial (poly vl pvl)
  parameterize-quadratic (poly vl pvl point)

- level4/

  - solve.lsp
    Function to find solutions to system of polynomial equations, using resultant calculations.

    solve (equations)

  - intersect.lsp
    Functions for intersecting implicit curves and surfaces, using solve.

    intersect2e2d (c1 c2)
    intersect2e3d (s1 s2)

  - output.lsp
    Functions for putting arguments into the protocol format for data transfer between C and Lisp.

    ganith-format (string args)
    ganith-format-string (s)
    CI-return (l)
    request-ui-computation (req args)

- ci.lsp
  C interface stub functions for each exported facility of the Computer Algebra library.

  resultant-CI (poly1 poly2 var)
  eliminate-CI (poly1 poly2 var)
  intersect2e2d-CI (c1 c2)
  intersect2e3d-CI (s1 s2)
  intersect3e3d-CI (s1 s2 s3)
  compactify-CI (spoly)
  param2dNv-CI (poly)
  realify2d-CI (poly)
  setvars-CI (v1 v2 v3)

- environ
  There are no functions defined in the files of this directory.

4.5 Extension Interfaces

Ganith provides simple ways to extend its capabilities. These extensions are in the form of **commands** that can be executed directly from the user-interface. A command always has the form *name(arg, ..., arg)*. The central idea behind the implementation of Ganith commands is simple: one writes a routine that performs some action, then a simple mechanism is used to link this routine, or *action procedure* to the command.

Whenever **Execute** is called, the text selection is matched against the pattern *func*(*args*). If it does, the *func* part is looked up in the table, and its corresponding action procedure is called with *args* (a string) as an argument. The action procedure may then interpret *args* in any way it chooses.

In this section, we describe the C and Lisp extension mechanisms.

4.5.1 C Language Interface

The information presented in this section should be enough to enable a programmer to rapidly add new functions to the Ganith user interface. The features described here are the action procedure linkage mechanism, and how to create graphical objects that can be displayed in the user interface.

Adding an Action Procedure. An action procedure must always take a string as an argument and return `void`. By convention, action procedures must have names ending with the letters "UI." Once an action procedure is defined, it must be declared in "actions.h" and have a one-line entry made in the data structure "FunctionTable" in that file, along with all the other action procedures. This entry consists of a pair (command name, action procedure name). This is all: once this is done, executing the command from the user interface will automatically invoke the corresponding action procedure.

For example, suppose one wishes to define a new command called "fancy." First, define a procedure as follows:

```
void
FancyUI(args)
char *args;
{
...
}
```

Then, add the following declaration in "actions.h":

```
extern void FancyUI();
```

and then insert the (name, procedure) pair into the look up table in the same file:

```
GanithFunct FunctionTable[] = {
   ...
   {''fancy'', FancyUI},
};
```

Finally, if "fancy" takes multiple arguments, there are utilities in "actions.c" to assist in command argument parsing; see the section "C Side Functions."

Creating Graphical Objects. To display something on the user interface, the standard way is to create a graphical object. Once a graphical object is created and entered into the global list of objects, it can be manipulated by the user in any window. Hence, it is suggested that any action procedure that wishes to do graphics follow the structure of the following code fragment:

```
void
FancyUI(args)
char *args;
{
   GrObject *pgo;
   ... /* argument processing and other computations */

   pgo = StartGrObjectDescription();
   ... /* any number of graphics calls */
   EndObjectDescription();

   /* add the object to the global object table */
   UIAddObject(pgo, ... );
}
```

4.5.2 Lisp Language Interface

In this section, we describe how how to create Lisp functions that can be called by the simulated foreign-function call facilities in the file "li.c," e.g. "CallLispFunction()."

Again, the syntax is very simple. Suppose one has a lisp function "fancy," that takes some lisp data structures as arguments and returns some lisp data structures, and wants to make this function accessible to C. Then, one must write a stub function named (by convention) "fancy-CI" that takes the same number of arguments as "fancy" except that each argument is a string. All that "fancy-CI" need do is to parse each argument into the appropriate format needed by "fancy" and then call "fancy." It must then convert each return value of "fancy" into a string and use the Lisp function "CI-return" to return these strings to C, where they can be accessed by "GetLispReturn()."

Thus the simulated foreign-function interface is similar in syntax to a real foreign-function interface, but it trades some efficiency for large gains in portability.

4.6 Building Ganith

For the latest instructions to build Ganith see the Ganith.README.Build file distributed with the source code. Building instructions for the current ganith source code distribution is as follows. It is assumed that the source code lies in a directory ".../src" that in turn contains two directories "ui" and "lisp," containing the C and Lisp parts respectively. Instructions for each part are described separately.

4.6.1 Building the User Interface

Throughout the user interface, Imakefiles are used instead of Makefiles. This allows some independence of directory locations for certain files such as X libraries. However, for various architectures, separate Imakefiles are still needed; it is just that the Imakefiles differ much less than corresponding Makefiles would. One could also maintain one complicated Imakefile with several targets, one for each architecture.

Building the user interface consists of building each sub-library (described earlier), and then building ganith from the ui directory. For each sub-library, enter its corresponding directory, type "xmkmf" followed by "make." It should build to completion.

When all sub-libraries are successfully compiled, the user interface binary is built likewise by typing "xmkmf" and "make." The appropriate Imakefile must be used; there are various files named "Imakefile.arch" where the suffix "arch" denotes the architecture.

A shell script "MakeAll" does all the above steps, so all one really needs to do is to install the correct Imakefile in the ui directory and run "MakeAll."

4.6.2 Building the Lisp Binary

The lisp binary must always reside in "../lisp/environ/ganith.lisp" relative to the "ui" directory. To build ganith.lisp, enter the "lisp/environ" directory and start a lisp process. Then load the file "make.lsp" which should convert each ".lsp" file into a ".o" file in each

of the levelN directories. Now, quit Lisp and start a fresh one. Load "init_ganith.lsp" to load all object files, and dump the running Lisp binary into a file "ganith.lisp."

The file "make.lsp" depends on a function "cd" (change directory) being defined. The "save binary" function will vary from Lisp to Lisp.

For Kyoto Common Lisp, the file "env.lsp" contains a definition of the "cd" command, and the "save binary" function is simply called "save." It takes the file name as an argument.

References

[1] S. Abhyankar and C. Bajaj. Automatic Rational Parameterization of Curves and Surfaces I: Conics and Conicoids. *Computer Aided Design*, 19(1):11–14, 1987.

[2] S. Abhyankar and C. Bajaj. Automatic Rational Parameterization of Curves and Surfaces II: Cubics and Cubicoids. *Computer Aided Design*, 19(9):499–502, 1987.

[3] S. Abhyankar and C. Bajaj. Automatic Rational Parameterization of Curves and Surfaces III: Algebraic Plane Curves. *Computer Aided Geometric Design*, 5(1):309–321, 1987.

[4] S. Abhyankar and C. Bajaj. Automatic Rational Parameterization of Curves and Surfaces IV: Algebraic Space Curves. *ACM Transactions on Graphics*, 8(4):324 – 333, 1989.

[5] V. Anupam, C. Bajaj, and A. Royappa. *The SHASTRA Distributed and Collaborative Geometric Design Environment.* Computer science technical report, capo-91-38, Purdue University, 1991.

[6] V. Anupam, C.Bajaj, A. Burnett, M. Fields, A. Royappa, and D. Schikore. *XS: A Hardware Independent Graphics and Windows Library* . Computer science technical report, capo-91-28, Purdue University, 1991.

[7] C. Bajaj. Local Parameterization, Implicitization and Inversion of Real Algebraic Curves. Computer science technical report, Purdue University, 1989.

[8] C. Bajaj. Geometric Computations with Algebraic Varieties of Bounded Degree. In Proc. of the Sixth ACM Symposium on Computational Geometry, pages 148–156, Berkeley, CA, 1990.

[9] C. Bajaj. Rational Hypersurface Display. Computer Graphics, 24(2):117–128, 1990.

[10] C. Bajaj. Surface fitting with implicit algebraic surface patches. In H. Hagen, editor, *Curve and Surface Modeling.* SIAM Publications, 1991.

[11] C. Bajaj, T. Garrity, and J. Warren. On the Applications of Multivariate Resultants. Computer science technical report, Purdue University, 1988.

[12] C. Bajaj, C. Hoffmann, J. Hopcroft, and R. Lynch. Tracing surface intersections. *Computer Aided Geometric Design*, 5:285–307, 1988.

[13] C. Bajaj and I. Ihm. Algebraic surface design with Hermite interpolation. Technical Report CSD-TR-939, Computer Sci., Purdue Univ., Jan. 1990. Revised in Dec. 1990. (To appear in ACM Transactions on Graphics).

[14] C. Bajaj and I. Ihm. C^1 smoothing of polyhedra with implicit surface patches. manuscript, 1991.

[15] C. Bajaj and R. Patterson. Curvature Adjusted Parameterizations of Curves. Csd-tr-907, Purdue University, 1989.

[16] J. Bloomenthal. Polygonization of Implicit Surfaces. Computer Aided Geometric Design, 5(00):341–355, 1988.

[17] J. Canny. Generalized Characteristic Polynomials, 1988.

[18] G. Collins. Subresultants and Reduced Polynomial Remainder Sequences. *JACM*, 14(1):128–142, 1967.

[19] G. Collins. The calculation of multivariate polynomial resultants. *JACM*, 18(4):515–522, 1971 (Oct.).

[20] F. Cucker, L. Pardo, M. Raimondo, T. Recio, and M. Roy. On the Computation of the Local and Global Analytic Branches of a Real Algebraic, 1989.

[21] Y. de Montaudouin and W. Tiller. Applications of Power Series in Computational Geometry. *Computer Aided Design*, 18(10):514–524, 1986.

[22] D. Duval. *Diverses questions relatives au calcul formel avec des nombres algébriques.* PhD thesis, L'Université Scientifique, Technologique et Médicale de Grenoble, 1987.

[23] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, 1987.

[24] R. Farouki. Concise Piecewise Linear Approximation of Algebraic Curves, 1988.

[25] A. Geisow. *Surface Interrogations.* PhD thesis, University of Anglia, School of computing Studies and Accountancy, 1983.

[26] P. Henrici. *Applied and Computational Complex Analysis*, 1988.

[27] I. Ihm. *Surface Design with Implicit Algebraic Surfaces.* PhD thesis, Purdue University, 1991.

[28] I. Ihm and B. Naylor. Piecewise linear approximations of digitized space curves with applications. In N.M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 545–569. Springer-Verlag, Tokyo, 1991.

[29] T. Sederberg and J. Snively. Parameterization of Cubic Algebraic Surfaces, 1987.